Application Note:
Heap Memory Management

Document Number: SWRA204

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
1.0	Initial release	12/06/2006
1.1	Changed document name. Updated title page	05/21/2007
1.2	Updated for ZStack-2.2.0 release	04/02/2009
1.3	Enhanced profiling and run-time efficiency with Long-Lived bucket.	09/20/2010

TABLE OF CONTENTS

i. Acronyms

API	Application Programming Interface
BSP	Board Support Package – taken together, HAL & OSAL comprise a rudimentary operating system commonly referred to as a BSP.
HAL	Hardware (H/W) Abstraction Layer
OSAL	Operating System (OS) Abstraction Layer
OTA	Over-The-Air

1. Heap Memory Management

1.1 Introduction

The OSAL heap memory manager provides a POSIX-like API for allocating and re-cycling dynamic heap memory. Two important considerations in a low-cost, resource-constrained embedded system, size and speed, have been duly addressed in the implementation of the heap memory manager.

- Overhead memory cost to manage each allocated block has been minimized – as little as *2 bytes* on CPU's with one- or two-byte-aligned memory access (e.g. 8051 SOC and MSP430).
- Interrupt latency for the allocation and free operations has been minimized – freeing is immediate with no computational load other than bounds checks and clearing a bit; allocating is very much sped-up with a packed long-lived memory block and a dynamically updated first-free pointer for high-frequency small-block allocations (e.g. OSAL Timers).

1.2 API

1.2.1 osal_mem_alloc()

The `osal_mem_alloc()` function is a request to the memory manager to reserve a block of the heap.

1.2.1.1 Prototype

```
void *osal_mem_alloc( uint16 size );
```

1.2.1.2 Parameters

`size` – the number of bytes of dynamic memory requested.

1.2.1.3 Return

If a big enough free block is found, the function returns a void pointer to the RAM location of the heap memory reserved for use. A NULL pointer is returned if there isn't enough memory to allocate. Any non-NULL pointer returned must be freed for re-use by invoking `osal_mem_free()`.[1]

1.2.2 osal_mem_free()

The `osal_mem_free()` function is a request to the memory manager to release a previously reserved block of the heap so that the memory can be re-used.

1.2.2.1 Prototype

```
void osal_mem_free( void *ptr );
```

1.2.2.2 Parameters

`ptr` – a pointer to the buffer to release for re-use – this pointer must be the non-NULL pointer that was returned by a previous call to `osal_mem_alloc()`.

1.2.2.3 Return

None.

1.3 Strategy

Memory management should strive to maintain contiguous free space in the heap, in as few blocks as possible, with each block as big as possible. Such a general strategy helps to ensure that requests for large memory blocks always succeed if the total heap size has been set properly for the application's use pattern.

The following specific strategies have been implemented:

- Memory allocation is not penalized by having to traverse long-lived heap allocations if the system initialization is implemented as recommended within this guide.
- Memory allocation for small-blocks almost always begins searching at the first free block in the heap.
- Memory allocation attempts to coalesce all contiguous free blocks traversed in an attempt to form a single free block large enough for an allocation request.
- Memory allocation uses the first free block encountered (or created by coalescing) that is big enough to meet the request; the memory block is split if it is usefully bigger than the requested allocation.

1.4 Discussion

It is immediately after system task initialization that the effective "start of the heap" mark is set to be the first free block. Since the memory manager always starts a "walk", looking for a large enough free block, from the aforementioned mark, it will **greatly** reduce the run-time overhead of the walk if all long-lived heap allocations are packed at the start of the heap so that they will not have to be traversed on every memory allocation. Therefore, any application should make all long-lived dynamic memory allocations in its respective system initialization routine (e.g. ZDApp_Init(),GenericApp_Init()). Within said system initialization routines, the long-lived items must be allocated before any short-lived items. Any short-lived items allocated must be freed before returning, otherwise the long-lived bucket may be fragmented and the run-time throughput adversely affected proportionally to the number of long-lived items that the osal_memory module is forced to iterate over *for every allocation* for the rest of the life of the system. As an example, if the system initialization function starts an OSAL Timer (osal_start_timerEx()), this may fragment the long-lived bucket because the memory allocated for the timer will be freed and re-used throughout the life of the system (even if coincidence happens that every free and re-use is simply for resetting the same timer.) The recommended solution in this case would be to set the event corresponding to the timer (osal_set_event ()) and then continue to restart the timer as appropriate in the application's event handle for the corresponding event (refer to the behavior of the hal_key polling timer and corresponding event, HAL_KEY_EVENT). On the other hand, a reload timer (osal_start_reload_timer()) is a long-lived allocation and is recommended to be started during system initialization of all other long-lived items.

The application implementer must ensure that their use of dynamic memory does not adversely affect the operation of the underlying layers of the Z-Stack. The Z-Stack is tested and qualified with sample applications that make minimal use of heap memory. Thus, the user application that uses significantly more heap than the sample applications, or the user application that is built with a smaller value set for `MAXMEMHEAP` than is set in the sample applications, may inadvertently starve the lower layers of the Z-Stack to the point that they cannot function effectively or at all. For example, an application could allocate so much dynamic memory that the underlying layers of the stack would be unable to allocate enough memory to send and/or receive any OTA messages – the device would not be seen to be participating OTA.

1.5 Configuration

1.5.1 MAXMEMHEAP

The `MAXMEMHEAP` constant is usually defined in `OnBoard.h`. It must be defined to be less than **`32768`**.

`MAXMEMHEAP` is the number of bytes of RAM that the memory manager will reserve for the heap – it cannot be changed dynamically at runtime – it must be defined at compile-time. If `MAXMEMHEAP` is defined to be greater than or equal to `32768`, a compiler error in `osal_memory.c` will trigger. `MAXMEMHEAP` does not reflect the total amount of dynamic memory that the user can expect to be usable because of the overhead cost per memory allocation.

1.5.2 OSALMEM_PROFILER

The OSALMEM_PROFILER constant is defined locally in osal_memory.c to be **FALSE** by default.

After the implementation of a user application is mature, the OSAL memory manager may need to be re-tuned in order to achieve optimal run-time performance with regard to the MAXMEMHEAP and OSALMEM_SMALL_BLKSZ constants defined. The code enabled by defining the OSALMEM_PROFILER constant to **TRUE** allows the user to gather the empirical, run-time results required to tune the memory manager for the application. The profiling code does the following.

1.5.2.1 OSALMEM_INIT

The OSALMEM_INIT constant is defined locally in osal_memory.c to be ascii '**X**'.

The memory manager initialization sets all of the bytes in the heap to the value of OSALMEM_INIT.

1.5.2.2 OSALMEM_ALOC

The OSALMEM_INIT constant is defined locally in osal_memory.c to be ascii '**A**'.

The user available bytes of any block allocated are set to the value of OSALMEM_ALOC.

1.5.2.3 OSALMEM_REIN

The OSALMEM_INIT constant is defined locally in osal_memory.c to be ascii '**F**'.

Whenever a block is freed, what had been the user available bytes are set to the value of OSALMEM_REIN.

1.5.2.4 OSALMEM_PROMAX

The OSALMEM_PROMAX constant is defined locally in osal_memory.c to be **8**.

OSALMEM_PROMAX is the number of different bucket sizes to profile. The bucket sizes are defined by an array:

```
static uint16 proCnt[OSALMEM_PROMAX] = { OSALMEM_SMALL_BLKSZ,
                                         48, 112, 176, 192, 224, 256, 65535 };
```

The bucket sizes profiled should be set according to the application being tuned, but the last bucket must always be 65535 as a catch-all. There are 3 metrics kept for each bucket.

- proCur – the current number of allocated blocks that fit in the corresponding bucket size.
- proMax – the maximum number of allocated blocks that corresponded to the bucket size at once.
- proTot – the total number of times that a block was allocated that corresponded to the bucket size.

In addition, there is a count kept of the total number of times that the part of heap reserved for "small blocks" was too full to allow a requested small-block allocation: proSmallBlkMiss.

1.5.3 OSALMEM_MIN_BLKSZ

The OSALMEM_MIN_BLKSZ constant is defined locally in osal_memory.c.

OSALMEM_MIN_BLKSZ is the minimum size in bytes of a block that is created by splitting a free block into two new blocks. The 1st new block is the size that is being requested in a memory allocation and it will be marked as in use. The 2nd block is whatever size is leftover and it will be marked as free. A larger number may result in significantly faster overall runtime of an application without necessitating any more or not very much more overall heap size. For example, if an application made a very large number of inter-mixed, short-lived memory allocations of 2 & 4 bytes each, the corresponding blocks would be 4 & 6 bytes each with overhead. The memory manager could spend a lot of time thrashing, as it were, repeatedly splitting and coalescing the same general area of the heap in order to accommodate the inter-mixed size requests.

1.5.4 OSALMEM_SMALL_BLKSZ

The OSALMEM_SMALL_BLKSZ constant is defined locally in osal_memory.c.

The heap memory use of the Z-Stack was profiled using the GenericApp Sample Application and it was empirically determined that the best worst-case average combined time for a memory allocation and free, during a heavy OTA load, can be achieved by splitting the free heap into two sections. The first section is reserved for allocations of smaller-sized blocks and the second section is used for larger-sized allocations as well as for smaller-sized allocations if and when the first section is full. OSALMEM_SMALL_BLKSZ is the maximum block size in bytes that can be allocated from the first section.

1.5.5 OSALMEM_SMALLBLK_BUCKET

The OSALMEM_SMALLBLK_BUCKET constant is locally defined in osal_memory.c.

OSALMEM_SMALLBLK_BUCKET is the number of bytes dedicated to the previously described first section of the heap which is reserved for smaller-sized blocks.

1.5.6 OSALMEM_NODEBUG

The `OSALMEM_NODEBUG` constant is locally defined in osal_memory.c to be **TRUE** by default.

The Z-Stack and Sample Applications do not misuse the heap memory API.[2] The onus to be equally correct is on the user application: in order to provide the minimum throughput latency possible, there are no run-time checks for correct use of the API. An application can be shown to be correct by defining the `OSALMEM_NODEBUG` constant to `FALSE`. Such a setting will enable code that traps on the following misuse scenarios.

- invoking `osal_mem_alloc()` with size equal to zero.

Warning: invoking `osal_mem_free()` with a dangling or invalid pointer cannot be detected.

1.5.7 OSALMEM_PROFILER_LL

The `OSALMEM_PROFILER_LL` constant is defined locally in osal_memory.c to be **FALSE** by default.

Normally, the allocations that are packed into the Long-Lived bucket by all of the system initialization should not be counted during "profiling" because they are not iterated over during run-time. But, in order to properly tune the size of the Long-Lived bucket for any given Application, this constant should be used for one run on the debugger with a mature implementation. The numbers used in the following example are for the 8051 SOC, GenericApp, with out-of-the-box settings and thus using this default:

```
#define OSALMEM_LL_BLKSZ           (OSALMEM_ROUND(417) + (19 * OSALMEM_HDRSZ))
```

1. Define OSALMEM_PROFILER and OSALMEM_PROFILER_LL to TRUE
2. Set a break point in osal_mem_kick() after this operation:
    i. ff1 = tmp - 1;
3. Inspect the variable 'proCur' in an IAR 'Watch' window and sum the counts of all of the buckets (19 in this example) and plug it into the formula above – this is the count of long-lived items.
4. Subtract the value of ff1 (0x1095 in one particular run) from the location of theHeap (0x0ECE in that same run) and then subtract the sub-total of the count of long-lived items multiplied by the OSALMEM_HDRSZ (19 * 2 = 38 for this example.)

Further memory profiling should now be done with OSALMEM_PROFILER_LL set back to FALSE so as not to count the long-lived allocations in the statistics.

Appendix A. Applicable Documents

Internal Documents

1. Z-Stack OSAL API , SWRA194
2. Z-Stack Compile Options, SWRA188

[1] There is no automatic garbage collection implemented in the memory manager. So any heap memory that is no longer used but not manually freed will be lost forever and is considered to be a "memory leak". The Z-Stack and Sample Applications are leak-free. A memory leak will eventually make the application appear to "lock-up" – eventually there will not even be enough free heap to set an OSAL timer or write a debug message to the serial port.

[2] Except the misuse noted below in section 1.4.6.

